OpenGL

“Because Its All Triangles”

What is OpenGL?

"The GL stands for Good Luck because you're gonna need it"

- OpenGL is mainly considered to be a graphics API widely used to render the graphics we see on the screen in many popular software.
- However, OpenGL by itself is not an API but merely a specification developed and maintained by the Khronos Group.
- The OpenGL specification specifies exactly what the result/output of each function should be and how it should perform.

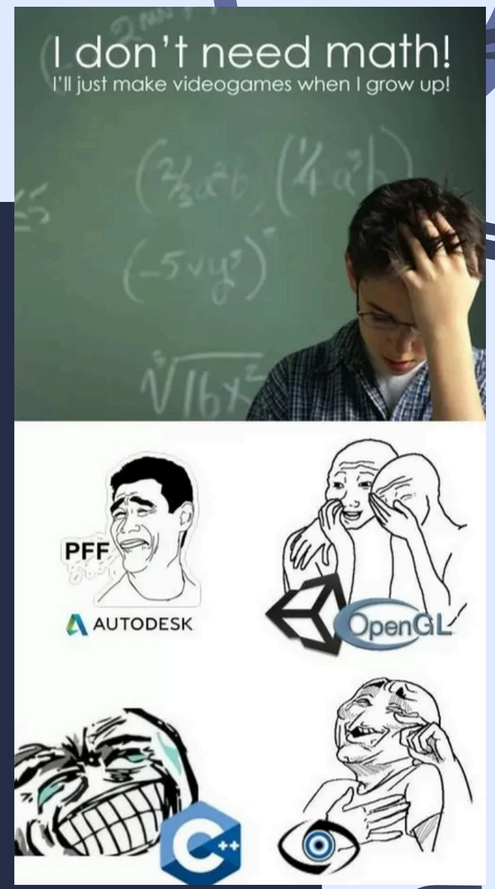
Who Makes It?

The Spanish Inquisition, unexpected, right?

- **Well, frankly, anyone who needs it.**
- **GPU manufacturers often tweak and create new libraries as extensions, in order for OpenGL functions to be compatible with the code**
- **This also means that whenever OpenGL is showing weird behavior that it shouldn't, this is most likely the fault of the graphics cards manufacturers.**

The World Revolves Around You

“When the math is Mathing”



No, Quite Literally

- Since OpenGL simulates everything from the POV of a Camera, in order to move or rotate or move an object, we apply some matrix transforms to their position vectors.
- The position of an object in the OpenGL frame is represented as a vector of 3 elements as:

$$\vec{v} = \begin{pmatrix} x \\ y \\ z \end{pmatrix}$$

- Sometimes, this is represented with an extra row set as 1 for easier calculations.

Transforms: Translation

This transform can be applied by algebraically adding to the matrices. This is represented as a matrix product as:

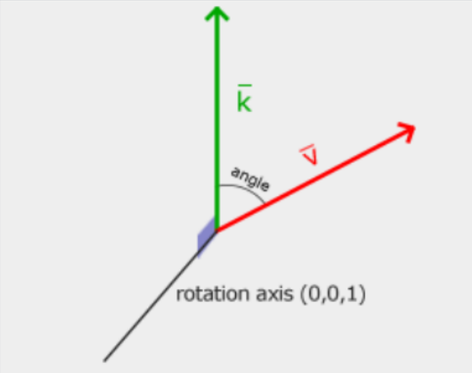
$$\begin{bmatrix} 1 & 0 & 0 & T_x \\ 0 & 1 & 0 & T_y \\ 0 & 0 & 1 & T_z \\ 0 & 0 & 0 & 1 \end{bmatrix} \cdot \begin{pmatrix} x \\ y \\ z \\ 1 \end{pmatrix} = \begin{pmatrix} x + T_x \\ y + T_y \\ z + T_z \\ 1 \end{pmatrix}$$

Hence, the transform matrix is obtained as :

$$\begin{bmatrix} 1 & 0 & 0 & T_x \\ 0 & 1 & 0 & T_y \\ 0 & 0 & 1 & T_z \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Transforms: Rotation

Using this axis system as reference:



We obtain, as a matrix product:

$$\begin{bmatrix} \cos\theta & -\sin\theta & 0 & 0 \\ \sin\theta & \cos\theta & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \cdot \begin{pmatrix} x \\ y \\ z \\ 1 \end{pmatrix} = \begin{pmatrix} \cos\theta \cdot x - \sin\theta \cdot y \\ \sin\theta \cdot x + \cos\theta \cdot y \\ z \\ 1 \end{pmatrix}$$

But Why Matrix Operations?

Are we all really a part of a simulation?

- Well, that is because matrix multiplications and operations allow us to perform multiple operations at the same time using Parallel computing.
- Since GPUs excel at parallel computing, they are well suited for these tasks.
- For Example: This is the time taken by GPU vs CPU(single core) for a random 10000 * 10000 matrix multiplication

```
Elapsed time is 0.003345 seconds.
```

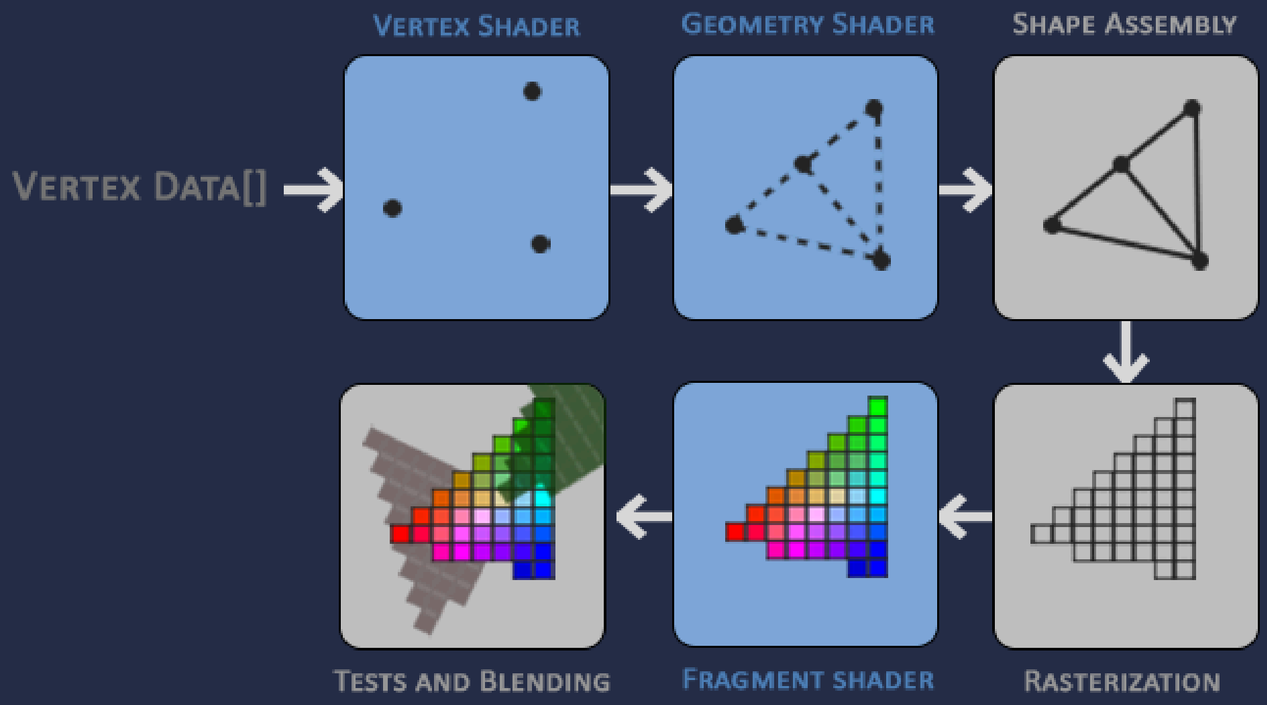
:Time taken by GPU

```
Elapsed time is 14.283674 seconds.
```

:Time taken by CPU

Render Pipeline

"I couldn't think of a punchline here, so I punched a pipe"



Render Pipeline

Vertex Shader

The vertex shader is defined, written and linked to the program

Geometry Shader

This is an optional step that is used to apply some geometric transforms to the output of the vertex shaders

Shape Assembly

In this step, the program takes as input all the vertices (or vertex if GL_POINTS is chosen) from the vertex (or geometry) shader that form one or more primitives and assembles all the point(s) in the primitive shape

Render Pipeline

Rasterization

The output of the primitive assembly stage is then passed on to the rasterization stage where it maps the resulting primitive(s) to the corresponding pixels on the final screen, resulting in fragments for the fragment shader to use.

Fragment Shader

This step calculates the color values of each pixel and maps the, to the associated pixel. Most OpenGL effects are applied in this step

Tests and Blending

This stage checks the fragment's corresponding depth (and alpha) value and uses those to check if the resulting fragment is in front or behind other objects and should be discarded(or blended) accordingly.

Time For Some Shady Stuff

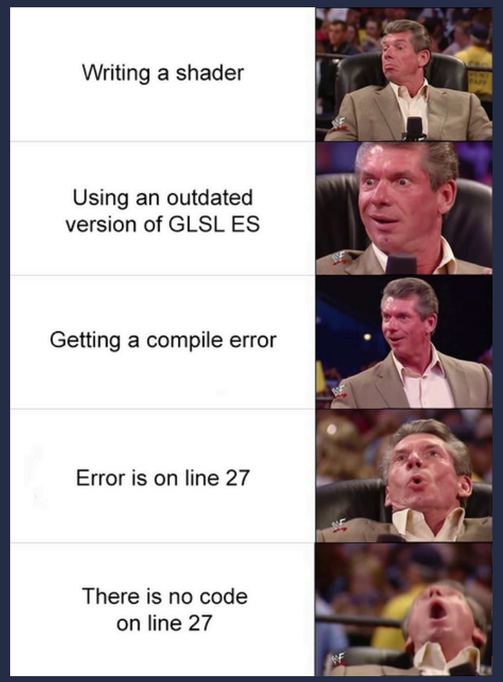
“Will the real Link Shader please stand up”

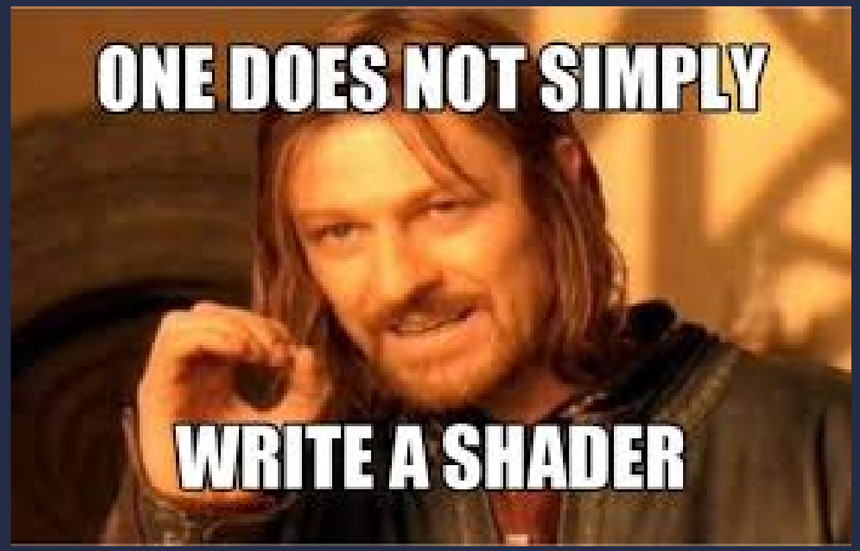
S(h)adistic OpenGL

“I know what I wrote”

- Shaders are little programs that rest on the GPU.
- These programs are run for each specific section of the graphics pipeline. In a basic sense, shaders are nothing more than programs transforming inputs to outputs.
- Shaders are also very isolated programs in that they're not allowed to communicate with each other
- The only communication they have is via their inputs and outputs.
- Its just that shaders don’t have the best reputation when it comes to developers.

Check out what I mean





But do not fret!
Once you learn it,
will not regret
On this note,
With this section let's get started.

Entonces, ¿qué lenguaje entienden los shaders?

“So what language do shaders understand?”

- Shaders are written in a C-like syntax, and are generally stored in the form of .txt files, which are then fed to the program by using standard file input and output.
- This syntax is called “GLSL”
- GLSL is tailored for use with graphics and contains useful features specifically targeted at vector and matrix manipulation
- It has libraries like “glm” that have pre-defined implementation of commonly used mathematical operations and objects.

GLSL Structure

"Get Shady and face the Aftermath"

Version and input variable declaration

In this part, we define the version number to be used for the shader and the input variables that need to be passed to the shader.

Definition of Output Variables

Here, we tell OpenGL what out shader will output, as that will be passed as input to the next shader in the pipeline

Declaration of other Variables, which are used in the code

In this part, we define **Uniforms,** which are used to send data from the CPU to the GPU. (The input variables send data across shaders). We can also define Helper Variables.

Main Function and Output Variables

Here, we define what operations need to be performed before assigning values to the output variables inside a main() function.

Accessing Shaders

Go Fetch! (the Shaders)

Creating a Window

We create a window using GLFW. This window is where all the graphics output shows up.

Making The Shader

We read the shader file and extract the text. We store it as a C string as OpenGL supports that format. We then store all the shaders in a sequence as each shader is characterised by its address, an unsigned int

Intermediates

This part varies from program to program. The most common implementation is usage of a Triangle Mesh which uses the following snippet to draw the triangle

```
glBindVertexArray(VAO);
glDrawArrays(GL_TRIANGLES, 0, vertex_count);
```

Clean Up

In this part, we delete the shader program we created and free the memory

Four Shade(r)s of OpenGL

NSFW Warning(excessive crying)

- Shaders are of (mainly) four types:
  a. Vertex Shaders
  b. Fragment Shaders
  c. Geometry Shaders
  d. Compute Shaders

- Additionally, there are two more types of shaders:
  - Tessellation Control Shaders
  - Tessellation Evaluation Shaders

Vertex Shaders

“You need to be sharp to implement these”

- The Vertex Shader is the programmable Shader stage in the rendering pipeline that handles the processing of individual vertices.
- Vertex shaders are fed Vertex Attribute data, as specified from a vertex array object by a drawing command.
- A vertex shader receives a single vertex from the vertex stream and generates a single vertex to the output vertex stream.
- There must be a 1:1 mapping from input vertices to output vertices.
- User-defined input values to vertex shaders are sometimes called "vertex attributes". Their values are provided by issuing a drawing command while an appropriate vertex array object is bound.
- Output variables from the vertex shader are passed to the next section of the pipeline.

Fragment Shaders

- A fragment shader is a stage in the OpenGL pipeline that processes fragments (potential pixels) generated during rasterization. It takes interpolated vertex data as input and outputs:
  - Depth value
  - Color values for the framebuffer
  - Stencil value (unmodified)
- The shader operates per fragment, allowing for operations like texturing and lighting, and is crucial for rendering detailed graphics.

Geometry Shaders

- A Geometry Shader (GS) is a Shader program written in GLSL that governs the processing of Primitives.
- The main reasons to use a GS are:
  - Layered rendering: taking one primitive and rendering it to multiple images without changing bound render targets.
  - Transform Feedback: This is often employed for computational tasks on the GPU (obviously pre-Compute Shader).

Compute Shaders

- A Compute Shader is a special type of shader in OpenGL designed for general-purpose parallel computing on the GPU.
- Unlike other shaders (vertex, fragment, geometry), compute shaders do not operate within the graphics pipeline—instead, they are executed as standalone programs for tasks like physics simulations, image processing, and AI.

Advanced Lighting

And then god said, “Let there be light”

- Normally, lighting in OpenGL is done by combining:
  a. Ambient Lighting
  b. Diffuse Lighting
  c. Specular Lighting
- A combination of these lighting methods gives us a pretty accurate depiction of the lighting in real life.
- This is called the ‘Phong’ lighting style.
- The main drawback with this method is that its specular reflections break down in certain conditions, specifically when the shininess property is low resulting in a large (rough) specular area.

Advanced Lighting

And then god said, “Let there be light”

You can see at the edges that the specular area is immediately cut off. The reason this happens is because the angle between the view and reflection vector doesn't go over 90 degrees. If the angle is larger than 90 degrees, the resulting dot product becomes negative and this results in a specular exponent of 0.0

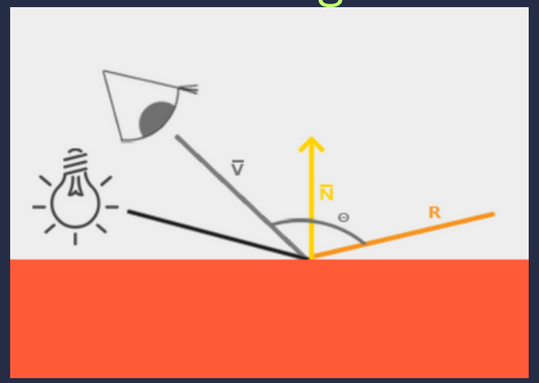
To overcome this problem, we introduce a halfway vector and use the “Blinn-Phong” lighting method

Blinn-Phong Lighting

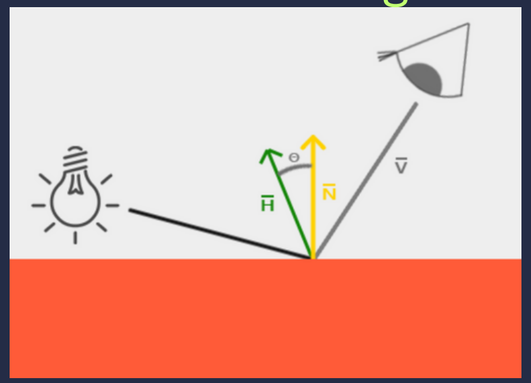
"I did not make this up I swear"

- In this method, instead of using the angle between the view vector and the normal vector as the Specular metric, we define a vector halfway between the view and the lighting vector and use the angle between the Halfway Vector and the Normal Vector as the Specular metric, ensuring that the angle never exceeds 90°.

Phong



Blinn-Phong



Where,

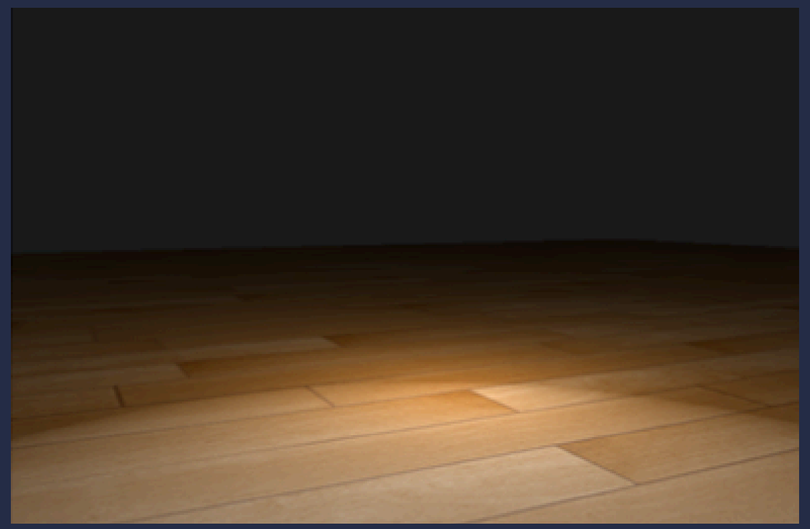
$$\bar{H} = \frac{\bar{L} + \bar{V}}{||\bar{L} + \bar{V}||}$$

Blinn-Phong Lighting

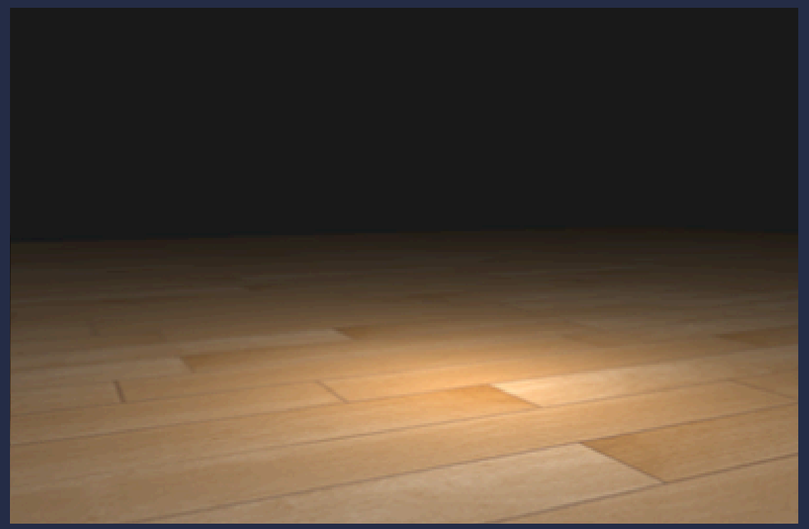
"I did not make this up I swear"

- An example on the previously used image:

Phong

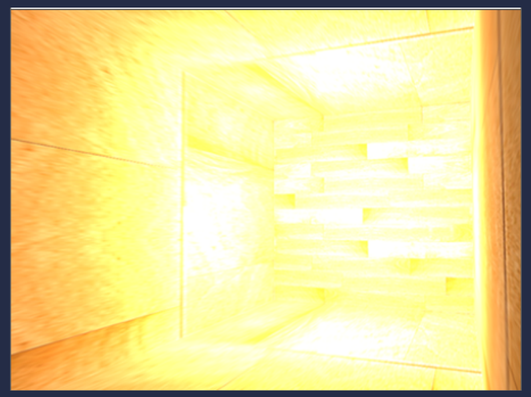


Blinn-Phong



HDR Lighting

And then god said, “Let there be better light”



When many pixel brightness intensities reach a value of more than 1, they are all clipped to 1. As a result, all those pixels appear at the same intensity, destroying the image quality. Resulting in an image like this.
One proposed solution is scaling down all pixels, which results in a sunglasses-like look

By allowing fragment colors to exceed 1.0 we have a much higher range of color values available to work in known as high dynamic range (HDR).

How it Works

“Let’s not oversaturate things”

- High dynamic range was originally only used for photography where a photographer takes multiple pictures of the same scene with varying exposure levels, capturing a large range of color values. Combining these forms a HDR image where a large range of details are visible based on the combined exposure levels, or a specific exposure it is viewed with.
- We allow for a much larger range of color values to render to, collecting a large range of dark and bright details of a scene, and at the end we transform all the HDR values back to the low dynamic range (LDR) of [0.0, 1.0].

We use tone-mapping algorithms like Reinhard tone mapping

Reinhard tone mapping

“Let’s not oversaturate things”

- Reinhard tone mapping involves dividing the entire HDR colour values into LDR colour values.
- The Reinhard tone mapping algorithm evenly balances out all brightness values onto LDR.

```
void main()
{
    const float gamma = 2.2;
    vec3 hdrColor = texture(hdrBuffer,
        TexCoords).rgb;
      // reinhard tone mapping
    vec3 mapped = hdrColor / (hdrColor +
                        vec3(1.0));
    FragColor = vec4(mapped, 1.0);
}
```

Result:

Vertex Post Processing

“Looks good to me, no bugs” -me, Friday evening

Transform Feedback’

Transform feedback is a way of recording values output from the Vertex Processing stage into Buffer Objects.

Clipping

Clipping involves removing the primitives that lie outside the view frustum to reduce computation complexity.

Perspective Division

Converts clip-space coordinates (x, y, z, w) to NDC (x/w, y/w, z/w)

Viewport Transformation

Maps NDC (-1 to 1 range) to screen coordinates (pixel space)

Transform Feedback

Ben 10 Nostalgia

- Transform Feedback is an OpenGL feature that captures the output of the vertex shader (or geometry shader) and writes it directly into a buffer without passing through the fragment stage.
- This allows us to store transformed vertices for later use, without sending them to the screen

The process involves:

1. Enable transform feedback mode
2. Bind a buffer to store the output
3. Define which variables to capture
4. Render geometry (without rasterization)
5. Retrieve transformed data

```
glEnable(GL_RASTERIZER_DISCARD);

glBindBufferBase(GL_TRANSFORM_FEEDBACK_BUFFER,
        0, feedbackBuffer);
glBeginTransformFeedback(GL_POINTS);

glDrawArrays(GL_POINTS, 0, NUM_VERTICES);

glEndTransformFeedback();
glDisable(GL_RASTERIZER_DISCARD);
```

Clipping

That reminds me, Clippy just got clipped out of existence, didn't he?

- Primitives generated by previous stages are collected and then clipped to the view volume. Each vertex has a clip-space position (the gl_Position output of the last Vertex Processing stage).
- While the clipping is automatically enabled by OpenGL, we can add an additional manual clipping factor by using:

```
float* gl_ClipDistance;
// This array is a built-in variable
glEnable(GL_CLIP_DISTANCEi);
// i corresponds to ith element
// of the gl_ClipDistance array
// gl_ClipDistance[i] > 0 implies
// ith point is inside
// the clipping volume
```

Perespective Division

Its a cultural divide, Imma get it on the floor

- The clip-space positions returned from the clipping stage are transformed into normalized device coordinates (NDC) via this equation:

$$\begin{pmatrix} x_{ndc} \\ y_{ndc} \\ z_{ndc} \end{pmatrix} = \begin{pmatrix} \frac{x_c}{w_c} \\ \frac{y_c}{w_c} \\ \frac{z_c}{w_c} \end{pmatrix}$$

Viewport Transform

- Converts NDC (-1 to 1 range) into window coordinates (pixel space).
- Ensures that objects are mapped correctly onto the frame buffer.

$$\begin{pmatrix} x_w \\ y_w \\ z_w \end{pmatrix} = \begin{pmatrix} \frac{width}{2} x_{ndc} + x + \frac{width}{2} \\ \frac{height}{2} y_{ndc} + y + \frac{height}{2} \\ \frac{farVal - nearVal}{2} z_{ndc} + \frac{farVal + nearVal}{2} \end{pmatrix}$$

Thank You